Winter 12-12-2004

Discovering Algebraic Specifications for Java Classes ; CU-CS-985-04

Johannes Henkel
University of Colorado Boulder

Christoph Reichenbach
University of Colorado Boulder

Amer Diwan
University of Colorado Boulder

Discovering Algebraic Specifications for Java Classes

University of Colorado Technical Report CU-CS-985-04*

Johannes Henkel and Christoph Reichenbach and Amer Diwan
Department of Computer Science, University of Colorado at Boulder

December 12, 2004

Abstract

Modern programs make extensive use of reusable software libraries. For example, a study of a number of large Java applications shows that between 17% and 30% of the classes in those applications use the container classes from the `java.util` package. Given this extensive code reuse in Java programs, it is important for the reusable interfaces to have clear and unambiguous documentation. Unfortunately, most documentation is expressed in English, and therefore does not always satisfy the above requirements. Worse yet, there is no way of checking that the documentation is consistent with the associated code. Formal specifications, such as algebraic specifications, do not suffer from these problems; however, they are notoriously hard to write.

To alleviate this difficulty we describe a tool that automatically derives documentation for interfaces of Java classes. Our tool probes Java classes by invoking them on dynamically generated tests and captures the information observed during their execution as algebraic axioms. While the tool is not complete or correct from a formal perspective we demonstrate that it significantly alleviates the task of writing formal documentation for reusable components.

1 Introduction

One of the hallmarks of modern programming languages is the support for reusable code. This support comes both in the form of language features (such as inheritance) and through extensive libraries for commonly used functionality and data structures (such as hash tables). Language support allows users of the language to create their own reusable libraries and to distribute these widely. Figure 1 gives

*Author's address: Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado 80309-0430, USA. This work is supported by NSF grants CCR-0085792, CCR-0133457, and CCR-0086255. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

	# classes	# using containers
Jedit 4.1	644	123 (19.1%)
Xalan J 2.5.2	2,395	398 (16.6%)
Jakarta Velocity 1.1b1	2,610	780 (29.9%)
Apache Tomcat 5.0.19	3,959	1,084 (27.4%)
Eclipse 3.0-M7	21,774	4,757 (21.8%)
JBoss 4.0.0DR2	32,820	7,888 (24.0%)

Figure 1: Use of containers based on Java container interfaces in typical Java programs

the number of classes in several Java applications that use the reusable container classes from the *java.util* package. We see that a significant fraction of classes in Java applications take advantage of these reusable containers.

Since Java applications rely so heavily on libraries, it is crucial for the libraries to be documented in a clear and unambiguous fashion. Moreover, it is desirable for the documentation to be at least somewhat machine checkable, so that library writers and users can automatically detect when documentation and code start to diverge. Today, most documentation for reusable code tends to be written in a natural language (e.g., English for the documentation of the Java standard libraries). As such, it is not machine checkable; moreover, it is even sometimes unclear, ambiguous, or incomplete. An alternative is to use formal specifications instead of a natural language to document code, since formal specifications are clear, unambiguous, and often machine checkable. However, formal specifications are difficult to write and debug. In prior work [HD04b] we described a system for helping programmers develop formal specifications. Here we describe a system that discovers formal specifications automatically.

In this work we focus on *algebraic specifications* [GH78], since they are particularly effective at describing the behavior of container classes, which, as discussed above, are heavily reused. Our system automatically discovers algebraic specifications from Java classes. Algebraic specifications can describe *what* Java classes implement without revealing implementation details.

Our approach is as follows: We start by extracting the *signatures* of classes automatically, using the Java reflection API. We then use the signatures to automatically generate terms, using heuristics to guide term generation. Each term corresponds to a legal sequence of method invocations on an instance of the class, i.e. to one that does not throw an exception. We then evaluate the terms and compare their outcomes. These comparisons yield equations between terms. Finally, we generalize these equations to axioms and use term rewriting to eliminate redundant axioms. We can increase confidence in the generated axioms by increasing the number of terms generated.

Our approach is inspired by recent work on discovering likely specifications based on program runs [ECGN01, ABL02, HL02, WML02]. Our methodology extends on this prior work in that it is the first to discover *high-level* functional specifications

that describe *how to use a component*. In contrast, prior systems, such as Daikon [ECGN01], discover Gries-style specifications that describe *how a component is implemented*.

We evaluate our system by running it on a number of container classes, including some from from the Java standard libraries. Our experiments reveal that our approach is effective in discovering specifications of Java classes. While we compare object representations as a run-time performance optimization, our results are not affected by any particular kind of object-internal representations and do not require any representational invariants. Since our approach is based on examining the run-time behavior of classes instead of static analyses, the axioms that it generates are not guaranteed to be correct. However, all but two of the axioms discovered by our system for the case studies were correct.

This paper improves and extends upon the presentation, ideas, and experimental results presented in our previous work [HD03].

The remainder of the paper is organized as follows: Section 2 discusses and illustrates the algebraic background of our specification language. Section 3 details our approach to dynamic specification discovery. Section 4 gives performance results and discusses two case studies for our system. Section 5 reviews related work, and Section 6 concludes.

2 Background

Our system discovers documentation for Java classes in a subset of the specification language described in our prior work [HD04b]. This specification language is designed to be as close as possible to the Java programming language, to make it easier for Java programmers to understand discovered specifications. This proximity to Java also means that the discovery tool can automatically map Java signatures and types to algebraic signatures and sorts.

To give a feel for the specification language, Figure 3 gives the specification in our language for the `ObjectStack` class in Figure 2. Algebraic specifications have two parts: an *algebraic signature* (e.g., lines 2-6 in Figure 3) and a set of *axioms* [Mit96] (e.g., lines 8-11 in Figure 3). The algebraic signature itself has two parts: *sorts* (e.g., lines 2-3 in Figure 3) and *operations* and their signatures (e.g., lines 4-6 in Figure 3). Intuitively, sorts give the types of interest to the underlying term algebra, whereas operations are the concrete entities from which terms in this algebra are built. The discovery tool uses reflection to trivially extract lines 1-7 of the specification from Java classes. The axioms equate terms in the algebra (e.g., lines 8-11 in Figure 3).

More precisely, given a Java method named `m` defined in a class represented by sort *cls* with n arguments $arg_1, \ldots, arg_n$ of sorts $sort(arg_1), \ldots, sort(arg_n)$, with the return type represented by sort *ret*, we construct the signature of an algebraic operation m within an algebra *cls* as follows:

$$\begin{aligned} m : cls \times sort(arg_1) \times ... \times sort(arg_n) \\ \rightarrow cls \times ret \times sort(arg_1) \times ... \times sort(arg_n) \end{aligned} \tag{1}$$

```
1  package edu.colorado.cs.simpleadts;
2
3  public class ObjectStack {
4      private Object [] store;
5      private int size;
6      private static final int INITIAL_CAPACITY=10;
7
8      public ObjectStack(){
9         this.store = new Object[INITIAL_CAPACITY];
10        this.size=0;
11     }
12
13     public void push(Object element){
14        if(this.size == this.store.length){
15           Object [] store = new Object[this.store.length*2];
16           System.arraycopy(this.store,0,store,0,this.size);
17           this.store = store;
18        }
19        this.store[this.size++]=element;
20     }
21
22     public Object pop(){
23        Object result = this.store[this.size];
24        this.store[this.size--] = null;
25        if(this.store.length > INITIAL_CAPACITY
26                 && this.size*2.7 < this.store.length){
27           Object [] store = new Object[this.store.length/2];
28           System.arraycopy(this.store,0,store,0,this.size);
29           this.store = store;
30        }
31        return result;
32     }
33 }
```

Figure 2: An object stack class implemented in Java.

```
1  specification ObjectStackSpecification                    ] spec. name
2  class ObjectStack is edu.colorado.cs.ObjectStack          ]
3  class Object is java.lang.Object                          ] sorts
4  method NewObjectStack is <void <init>()>                  ]
5  method push is <void push(java.lang.Object)>              ]
6  method pop is <javal.lang.Object pop()>                   ] operations
7  define ObjectStack                                        ] simulation set
8  forall s:ObjectStack forall o:Object       (Axiom 1)      ]
9        pop(push(s, o).state).retval == o                   ]
10 forall s:ObjectStack forall o:Object       (Axiom 2)      ]
11       pop(push(s, o).state).state == s                    ] algebraic axioms
```

Figure 3: Example Specification for an `ObjectStack` class (see Fig. 2).

The receiver argument (of sort *cls*) to the left of the arrow characterizes the original state passed into m. The receiver type to the right of the arrow represents the possibly modified state of the receiver as a result of evaluating the operation. The right hand side of the arrow also includes the return value (of sort *ret*, which can be the sort `void`). In case the operation corresponds to a Java constructor, the receiver argument to the left of the arrow is of type `void`. Note that while our full specification language [HD04b] can capture side effects to arguments, this paper assumes that invoking a method may only modify the observable state of the receiver (of type *cls*) and return a result (of type *ret*, cf. equation 1). Our system can be extended to handle other kinds of side effects at the cost of longer execution times.

Consider the two axioms in lines 8-11 in Figure 3. The `.retval` and `.state` qualifications select elements from the result tuple. `.state` retrieves the first element, which is the possibly modified receiver object (`this`). `.retval` retrieves the second element, which is the method's return value. Both axioms are universally quantified over all object stacks and all objects. Axiom 1 (Figure 3) states that invoking `pop` after a `push` returns the object that was last pushed. Axiom 2 states that invoking `pop` on an `ObjectStack` right after invoking `push` reverts the stack back into its prior, pre-push state.

These axioms concisely and clearly document the interface of the *ObjectStack*.

3 Discovering Algebraic Specifications from Java Classes

We discover algebraic specifications automatically from Java classes. The discovered specifications use the notation illustrated in Section 2.

Like other specification discovery tools (e.g., [Ern00]), our tool is based on dynamic program analysis. It generates sequences of invocations on a given class, and asks the JVM to evaluate them. Based on the results of the evaluation, the discovery tool constructs equations, which it generalizes to yield axioms (such as axioms 1 and 2 in Fig. 3). Unfortunately, running all test cases needed to ensure that a generalization is sound is often impossible or at least impractical. Therefore, our discovery tool does not guarantee soundness but instead focuses on providing a good starting point for writing a correct and complete specification for an interface. Our discovery tool works well with our specification interpreter [HD04b], which is useful in completing the specification produced by the discovery tool.

Fig. 4 gives an overview of our approach for discovering algebraic specifications automatically from Java classes. We start by generating terms (Sections 3.1 and 3.2). The terms consist of operation applications (which correspond to Java method and constructor invocations) and constants, such as integers and particular Java objects. We use these terms to generate equations (Section 3.4). For example, we may find that two terms evaluate to the same value. We use *observational equivalence* [DF94] to determine if values produced by two terms are equal (Section 3.3): We consider two values the same if they behave the same with respect to their

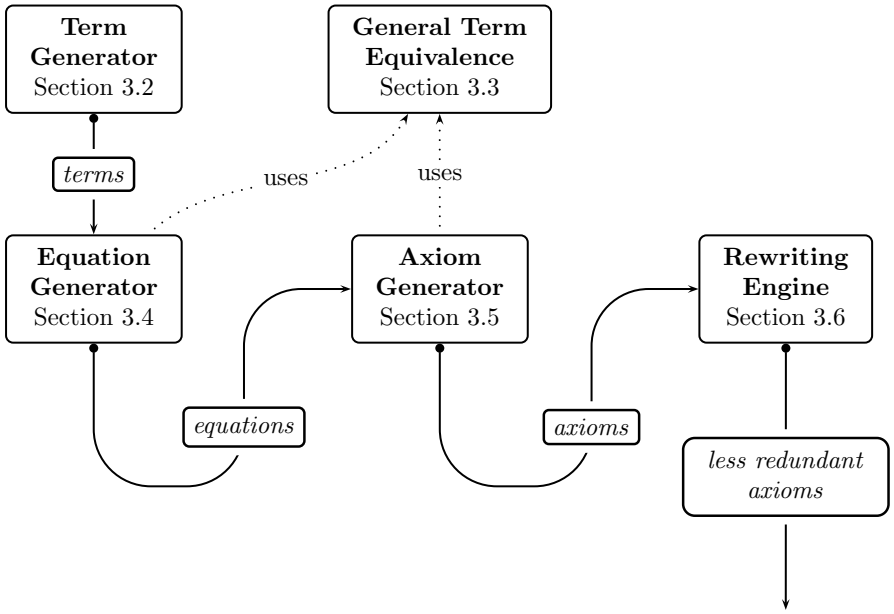


Figure 4: Architectural overview of our tool for discovering algebraic specifications

public methods. Thus, observational equivalence abstracts away from low-level implementation details and only considers the interface. We generalize the equations into axioms by replacing subterms with universally-quantified typed variables (Section 3.5). To determine (with some probability) whether the axioms hold, we check them by generating test cases. Finally, we rewrite our axioms to eliminate many redundant axioms (Section 3.6).

3.1 Automatically Mapping Java Classes to Algebras

Due to the design of our specification language (Section 2), the mapping from Java classes to algebraic signatures is trivial: each Java type is a sort in the algebra and each Java method is an operation on the appropriate sorts. For example, our system discovers the sorts and operations declared in lines 2-6 in Fig. 3 automatically from the `ObjectStack` class (Fig. 2): The system queries the Java reflection API at runtime to find the methods that belong to the input class and the argument and return types of the methods.

3.2 Generating Terms

The *term generator* (often abbreviated as just "generator" in this section) emits an endless stream of terms (see Figure 4). Each term models a particular object state. For example, the following two terms both model the state of an empty

object stack.

```
NewObjectStack().state
pop(push(NewObjectStack().state, Object@7).state).state
```

`Object@7` refers to a particular instance of the `Object` class.

Terms serve the role of test cases in our discovery tool. Analogously to a test-case generator, the goal of the generator is to produce terms that thoroughly explore the state space of the input class. Thus, the generator produces terms that end with `.state` and not with `.retval`.

3.2.1 Growing Terms

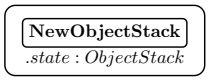
To generate a term for a particular algebra, the term generator first constructs an operation application corresponding to a Java constructor invocation. For example, to generate a term for the ObjectStack algebra the term generator starts with NewObjectStack, as shown in Fig. 5*a*. Then, the term generator grows the term incrementally, by adding one operation application (corresponding to a Java method invocation) at a time. The term generator backtracks whenever the addition of an operation application causes the term to generate an exception when executed. In the example in Fig. 5, the term generator backtracks (to step *c*), since the pop operation applied to an empty stack yields an exception (step *b*).

The generator needs to provide arguments for the operations used to grow the term. From a conceptual point of view, it could simply recurse to generate arguments of the appropriate type. However, for efficiency and quality of ultimate results, we identify two special cases that we handle differently: arguments of a primitive type (such as integer) and arguments that are instances of immutable classes (i.e., classes, such as Object, that do not have any modifying operations).

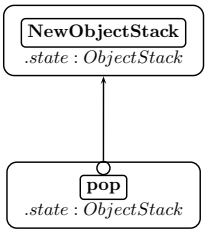
Arguments of primitive types For arguments of a primitive type, we use values from a fixed set (e.g., values between 0 and 10). If our set is too small, we will not explore enough of the state space and may end up missing some axioms; also, we may fail to invalidate proposed axioms for which a counter-example exists. If the set is too large, the tool will become too inefficient for practical use (we generate all terms up to a given size). In our experience, our approach works well for types such as integers and booleans. We have no experience with floating point types.

Arguments of immutable classes An *immutable class* is a class that has no methods for modifying the state of an instance after creation (e.g., Object and String classes in Java). Whenever the generator needs to produce an argument of an immutable class, it picks a value from a fixed set of instances of that class. For example, the generator may produce the term $push(ObjStack().state, \text{obj@4}).state$ where obj@4 is an instance from the precomputed set. As with primitive types, it is important for the set to be neither too small nor too large.

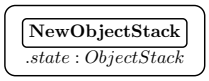
For non-immutable classes, we can not safely replace subterms with a particular instance as a precomputed constant, since then the potentially modified state



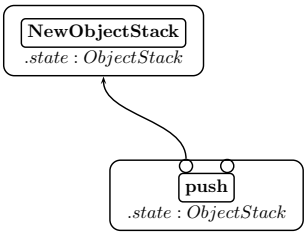
a. $NewObjectStack().state$: Executes fine.



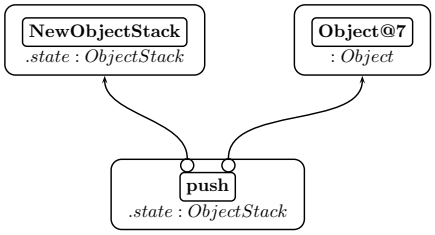
b. $pop(NewObjectStack().state).state$: Throws exception.



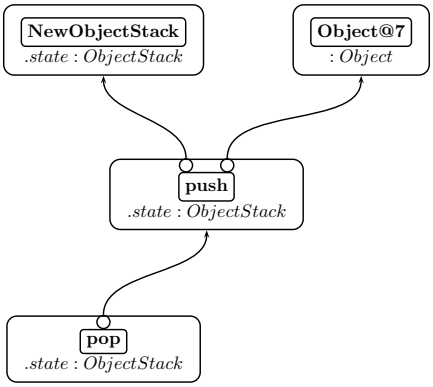
c. Backtracking step.



d. $push(ObjectStack().state, ?).state$: Argument is missing.



e. $push(NewObjectStack().state, Object@7).state$: Executes fine.



f. $pop(push(NewObjectStack().state, Object@7).state).state$: Executes fine.

Figure 5: Growing terms incrementally.

of the instance would be used in the next evaluation of the term. For example, if we execute the term $push(\text{ObjectStack@4}, obj@4).state$ multiple times, we will compute a different inner state for the receiver each time.

While algorithmic approaches to determining class immutability automatically are conceivable, enumerated immutable types manually for our experiments, since we observed better results with the latter approach. A third possible approach would have been the use of a mod-ref analysis, which we expect to explore in future work.

3.2.2 Policy for Term Generation

As discussed above, the quality of the terms ultimately determines the quality of the axioms discovered by our system. If we generate too few terms, then we may end up with incorrect axioms; if we generate too many terms, the axiom discovery may take a long time. Our current system generates all terms up to a user-defined size. The user configures the term sizes used for discovery and test generation for each class that is being discovered (for examples of term sizes used in our experiments, see Section 4.1).

3.2.3 Optimizations

To make the exploration of large object state spaces practical, we employ optimizations to discard redundant terms. Terms are redundant if they do not help us in learning more about the behavior of objects.

Optimization 1: Avoid side-effect free operations Since the generator needs to explore the state space of instances of a class, operations that do not modify the state of the receiver need not be part of any terms. For example, the `toString()` method usually does not change the state of the receiver. Thus, the generator does not consider the operation `toString` when generating terms.

The generator determines whether applying a particular operation would change the state of an object by (i) executing the term to generate an object, (ii) saving the state of the object (e.g., by serializing it), (iii) applying the operation to the object, and (iv) comparing the state of the object with the saved state. For example, to determine if `toString` modifies the state of the receiver, the generator will serialize the receiver before and after applying `toString` and compare the two serialized versions. The generator will find the two to be the same and will thus conclude that `toString` does not modify the internal state of the receiver.

This technique is conservative: Whenever it indicates that an operation application is redundant, it will be right; however, it cannot find all possible redundancies, since the serialized inner state being different does not guarantee that the objects before and after the applications are different in any externally observable way.

Optimization 2: Only generate the minimal term for any generated object representation We can understand terms as recipes for generating particu-

lar Java objects. When two different terms generate objects with the same internal representation, the generator keeps only the smaller of the two terms.

Whenever the generator produces a term it executes it and attempts to put a serialization of the resulting state into a hash table. If there is a conflict in the hash table, the generator knows that it has seen the state before and thus the new term is unnecessary. Since the generator produces terms in order of increasing size, this mechanism will always discard the larger terms in favor of smaller terms. The equation generator (Section 3.4) also uses the conflicts in the hash table to cheaply identify equations.

3.3 Term Equivalence

To determine whether or not two terms are the same, the specification discovery tool uses a variation of Doong and Frankl's EQN method [DF94]. Doong and Frankl define that two values are *observationally equivalent* if they behave the same when we apply arbitrary operations to both of them. Two values can be observationally equivalent even though they have different internal representations. For illustration, consider an ObjectStack implementation in Java like the one shown in Fig. 2, except that line 25 is missing. For this implementation, the terms

```
ObjectStack().state
```

and

```
pop(push(ObjectStack().state, Object@7).state).state
```

generate two objects with a different representation: The first term generates an `ObjectStack` instance with

$$size = 0 \ \wedge \ store.length = 10 \ \wedge \ \forall i \ . \ 0 \leq i < 10 \Rightarrow store[i] = null$$

For the second term, $store[0]$ contains a reference to an object, since the `pop` operation fails to fill the unused slots with `null`, thereby creating a leak. Even though the two example terms generate a different object representation, both objects are still observationally equivalent.

We extend Doong and Frankl's notion of observational equivalence by introducing the concept of consistency, which allows us to accommodate Java methods such as `hashCode` (Section 3.3.1), and we exploit reference equality for performance optimizations (Section 3.3.2).

The EQN method for observational equivalence is effective but inefficient since it needs to apply many operations to both terms in order to gain confidence that they are indeed the same. Thus, we identify three special cases that we can handle quickly: primitive types, immutable classes, and representation equivalence. Algorithm 1 describes how we dispatch between the four possibilities. In Algorithm 1, the parameters for GTE-DISPATCH are terms, not the values computed by the terms. This is necessary since each of PRIMITIVE-EQUALS, REFEQ-APPLIES, REP-EQUALS, and EQN needs more than a single sample of what the terms evaluate to. These functions have the following meanings:

Algorithm 1 General term equivalence dispatch

```
GTE-DISPATCH(t1, t2) begin
Require: t1, t2 are terms of the same type T.
  if T is a primitive type then
    return PRIMITIVE-EQUALS(t1,t2)
  else
    if REFEQ-APPLIES(t1, t2) then
      return EVAL(t1)=ref EVAL(t2)
    else
      if REP-EQUALS(t1, t2) then
        return true
      else
        return EQN(t1,t2)
      end if
    end if
  end if
end
```

- EVAL(t) denotes the result of the evaluation of t
- $=_{ref}$ checks reference equality
- PRIMITIVE-EQUALS determines equality among primitive types (section 3.3.1)
- REFEQ-APPLIES determines equality of references computed by terms (section 3.3.2)
- REP-EQUALS decides representational equivalence (section 3.3.3)
- EQN computes whether observational equivalence can be assumed (section 3.3.4)

GTE-DISPATCH is conservative in that it is accurate whenever it exposes non-equivalence. It may be inaccurate when it finds equivalence.

We now describe the algorithms for the four cases in more detail.

3.3.1 Equivalence for Primitive Types

Let's suppose the `ObjectStack` had a `size` method. We can confirm the equivalence

$$size(push(ObjectStack().state, Object().state).state).retval = 1$$

by observing that the evaluation of the corresponding Java invocation sequences yields 1.

Now consider the term $hashCode(ObjectStack().state).retval$. Since the *hashCode* function will compute a different value for each *ObjectStack* instance,

Algorithm 2 Equivalence for primitive types

PRIMITIVE-EQUALS(t_1, t_2) **begin**
Require: t_1, t_2 are terms computing values of the same primitive type.
  $result_{1a} \leftarrow$ EVAL(t_1), $result_{1b} \leftarrow$ EVAL(t_1)
  $result_{2a} \leftarrow$ EVAL(t_2), $result_{2b} \leftarrow$ EVAL(t_2)
  $consistent_1 \leftarrow result_{1a} =_{val} result_{1b}$
  $consistent_2 \leftarrow result_{2a} =_{val} result_{2b}$
  if not $consistent_1$ **and not** $consistent_2$ **then**
    return true
  end if
  if not $consistent_1$ **or not** $consistent_2$ **then**
    return false
  end if
  return $result_{1a} =_{val} result_{2a}$
end

the term will evaluate to a different value each time. We also say that this term is not *consistent*; A *consistent* term produces the same result each time it is executed. To approximate consistency, Algorithm 2 evaluates each term twice. If both evaluations yield the same result, we consider the term as consistent. If neither term is consistent, we consider them equal. If exactly one term is consistent, we consider them to be non-equal. Finally, if both terms are consistent, we can compare them by determining their value equality.

3.3.2 Comparing the References Computed by Terms

Algorithm 3 Checking for reference equality

REFEQ-APPLIES(t_1, t_2) **begin**
Require: t_1, t_2 are terms of the same reference type.
  return EVAL(t_1) $=_{ref}$ EVAL(t_1)
    and EVAL(t_2) $=_{ref}$ EVAL(t_2)
end

Since our algorithm currently handles only side effects to instance variables of receivers and we use instances of immutable classes from a fixed set (Section 3.2), there are many situations where we can use reference equality rather than resorting to the more expensive observational equivalence algorithm (Section 3.3.4). Thus, we use a heuristic similar to that for primitive types (Algorithm 3): we evaluate each term twice and see if each term evaluates to the same value in both evaluations. If they do, then we can simply compare the references that they return. If they do not, then we use the observational equivalence procedure. For example, for the terms $pop(push(ObjStack().state, \text{obj@123}).retval$ and obj@123, REFEQ-APPLIES

returns true, while it returns false for

$$pop(push(ObjStack().state, \text{obj@123}).state$$

and $ObjStack().state$.

3.3.3 Representation Equivalence

Algorithm 4 Representation equivalence

REP-EQUALS(t_1, t_2)
Require: t_1, t_2 are terms, evaluating to instances of some class C
  if SERIALIZE(EVAL(t_1))=$_{val}$SERIALIZE(EVAL(t_2)) **then**
    return true
  else
    return false
  end if

Algorithm 4 shows pseudocode for a test of representation equivalence. Representation equivalence implies observational equivalence, which is why we use this check as an optimization. Representation equivalence checks if both terms evaluate to objects with identical inner state by using SERIALIZE which serializes an object into a binary representation, such as a `byte` array. If the objects have the same inner state, we can safely assume that they are observationally equivalent.

3.3.4 Observational Equivalence

Algorithm 5 Observational equivalence

EQN(t_1, t_2)
Require: t_1, t_2 are terms, evaluating to instances of some class C
  repeat
    Generate a test stub *stb*, with an argument of type C
    for all observers *ob* applicable to evaluation results of *stb* **do**
      $stubapp_1 \leftarrow stb(t_1)$, $stubapp_2 \leftarrow stb(t_2)$
      $obsapp_1 \leftarrow ob(stubapp_1, \ldots).retval$, $obsapp_2 \leftarrow ob(stubapp_2, \ldots).retval$
      if not GTE-DISPATCH($obsapp_1$,$obsapp_2$) **then**
        return false
      end if
    end for
  until confident of outcome
  return true

Algorithm 5 shows pseudocode for our version of EQN. EQN approximates observational equivalence of two terms of class C as follows. We start by generating term stubs that take an argument of type C and apply the stubs to t_1 and t_2. We

then pick an observer[1] and apply it to both terms. We compare the outputs of the observers using GTE-DISPATCH (Algorithm 1). If GTE-DISPATCH returns false, then we know that t_1 and t_2 are not observationally equivalent, otherwise we become more confident in their equivalence. Since this algorithm could potentially run into an infinite recursion, we use a recursion limit (omitted for brevity).

For example, consider applying this procedure to terms from the *ObjectStack* algebra. An example of a stub is $\lambda x.push(x, obj@2).state$ and an example of an observer is *pop*. If t_1 is $push(ObjectStack(), obj@3).state$, the application of the stub and the observer yields

$$pop(push(push(IntStack().state, obj@3).state, obj@2).state).retval$$

In this particular example, recursing to GTE-DISPATCH will return equality for any t_2, as the stub dominates the observations gathered from the observer; a difference between t_1 and t_2 would be caught only after trying a different stub (such as $\lambda x.x$, the identity function) or observer (not possible here, as *pop* is the sole observer of this algebra).

3.4 Finding Equations

The form of the equations determines the form of the algebraic specifications that our system will discover. Our current implementation handles only equalities.

We can easily add new types of equations and can enable or disable equation types. So far we have added three kinds of equations to our implementation: state equations (Section 3.4.1), observer equations (Section 3.4.2), and difference equations (Section 3.4.3).

3.4.1 State Equations: Equality of Distinct Terms

For example,

$$pop(push(ObjectStack().state, obj@4).state).state = ObjectStack().state$$

is a state equation. These equations are useful in characterizing how operations affect the observable state of an object. We generate these equations whenever we find that two distinct terms are equivalent. We get some of these equations from "Optimization 2" in Section 3.2.3: whenever we have a conflict in the hash table, we have a potential equation. Since hashing does not use full observational equivalence, this method will only find some of the state equations. We call the optimized equation generator state/hash, while the more general generator (using observational equivalence) is called state/eqn.

3.4.2 Observer Equations: Equality of a Term to a Constant

These equations take the following form (where *obs* is an observer and c a constant):

[1] An observer is an operation *op* such that the type of $op(\dots).retval$ is not void.

$$obs(term_1, arg_2, ..., arg_n).retval = \text{c}$$

Observer equations characterize the interactions between operations that modify and operations that observe. For example,

$$size(push(ObjectStack().state, obj@4).state).retval = 1$$

is an observer equation.

To generate observer equations we start with a term and apply an operation that returns a constant. We execute the term using the Java Reflection API, and record the evaluation result. We make sure that the term consistently returns the same constant. We then form an equation by equating the term to the constant. Note that constants might be object constants. This is where the scheme for immutable objects described Section 3.2 ("Arguments of immutable classes") becomes important: During term generation, we use immutable objects as constants; often, those objects then become the return value of a term. For example,

```
pop(push(ObjectStack().state,Object@1).state).retval
```

evaluates to the object constant `Object@1`.

3.4.3 Difference Equations: Constant Difference Between Terms

These equations take the following form (where *obs* is an observer, *op* an operation computing a value of a primitive type, and *diff* a constant of a primitive type):

$$op(obs(term_1.state).retval, diff).retval = obs(term_2).retval$$

For example:

$$IntAdd(size(ObjectStack().state), 1).retval$$
$$= size(push(ObjectStack().state, obj@3).state).retval$$

In this example, *op* is *IntAdd* (i.e., integer addition) and *diff* is 1.

To generate such axioms, we generate two terms, apply an observer to both terms, and take their difference. In practice, we found that difference equations with a small value for *diff* are the most interesting ones. Therefore, we only generate such an equation if *diff* is lower than a fixed threshold. This technique filters out most spurious difference equations.

3.5 Generating Axioms

Our axioms are 3-tuples (t_1, t_2, V), where t_1 and t_2 are terms and V is a set of universally-quantified, typed variables that appear as free variables in t_1 and t_2. An equation is simply an axiom with $V = \{\}$.

The generation of axioms is an abstraction process that introduces free variables into equations. For example, given the equation

$$\begin{aligned} &IntAdd(size(ObjectStack().state).retval, obj@1).retval \\ &= size(push(ObjectStack().state, obj@3).state).retval \end{aligned} \tag{2}$$

our axiom generator can abstract $ObjectStack().state$ into the quantified variable s of type ObjectStack and $obj@3$ into i of type `Object` to discover the axiom

$$\begin{aligned} &\forall s : ObjectStack \; \forall i : Object \\ &\quad IntAdd(size(s).retval, 1).retval = size(push(s, i).state).retval \end{aligned} \tag{3}$$

Algorithm 6 Axiom generation

```
generateAxiom(Algebra) begin
  (term1, term2) ← generate an equation in Algebra
  Subterms ← the set of subterms occurring in term1 or term2
  V ← a set of typed, universally-quantified variables x1, ..., xn
        with one xi for each subtermi ∈ Subterms
  for xi ∈ V do
    Replace each occurrence of subtermi with the variable xi in term1 and term2
    Generate a large set of test cases testset where each test case is a set of
          generated terms {test1, ..., testn}, such that testj can replace xj ∈ V
    for testcase ∈ testset do
      Set all xj ∈ V to the corresponding testj ∈ testcase
      if EQN-DISPATCH(term1,term2)= false then
        Undo the replacement of subtermi in term1 and term2
        Stop executing test cases
      end if
    end for
  end for
  Eliminate all xi from V which occur neither in term1 nor in term2
  return the axiom (term1, term2, V)
 end
```

Algorithm 6 describes the axiom generator, with optimizations left out for clarity. To generate an axiom for a particular algebra, we first use any of the equation generators as described in Section 3.4 to come up with an equation. We then compute the set of all subterms of term_1 and term_2. For example, given the terms $push(ObjectStack().state, 4).state$ and $ObjectStack().state$, the set of subterms would be $\{push(ObjectStack().state, obj@4).state, \; ObjectStack().state, \; obj@4\}$. We then initialize V as the set of universally-quantified variables, so that for each subterm there is exactly one corresponding universally-quantified variable in V. The loop then checks for each subterm, whether we can abstract all occurrences to a free variable. First, we replace all occurrences of the subterm with a free variable. Then, we generate test cases, where each test case replaces all the free variables in

the terms with generated terms. We compare whether $term_1$ and $term_2$ are equivalent under all test cases, and if a reasonable number of test cases is available. If not, we undo the replacement of the particular subterm. Note that our notion of a "reasonable number of test cases" is currently somewhat ad-hoc: Our system is currently hard-wired to require at least two test cases; we expect to provide more sophisticated facilities based on confidence metrics in the future.

At the end, we eliminate all free variables that do not occur in the terms and return the axiom.

3.6 Axiom Redundancy Elimination by Axiom Rewriting

The axiom generator (Section 3.5) generates many redundant axioms. For example, for the *ObjectStack* algebra, our generator may generate both

$$\begin{array}{l} \forall x_4 : \mathit{ObjectStack}, \forall i_4, j_4 : \mathit{Object} \\ \mathit{pop}(\mathit{pop}(\mathit{push}(\mathit{push}(x_4, i_4).\mathit{state}, j_4).\mathit{state}).\mathit{state}).\mathit{state} = x_4 \end{array} \tag{4}$$

and

$$\forall x_5 : \mathit{ObjectStack}\ \forall i_5 : \mathit{Object}\ .\ \mathit{pop}(\mathit{push}(x_5, i_5).\mathit{state}).\mathit{state} = x_5 \tag{5}$$

We eliminate redundant axioms using term rewriting [Mit96]. We use axioms that satisfy these two requirements as *rewriting rules*: (i) the left and right-hand sides must be of different length; and (ii) the free variables occurring in the shorter side must be a subset of the free variables occurring in the longer side. When using a rewrite rule on an axiom, we try to unify the longer side of the rewrite rule with terms in the axiom. If there is a match, we replace the term with the shorter side of the rewrite rule.

Whenever we are about to add a new axiom we note if any of the existing rewrite rules can simplify the axiom. If the simplified axiom is a rewrite rule we try to rewrite all existing axioms with this rule. If the rewriting makes an axiom redundant or trivial we throw it away. If a rewritten axiom yields a rewrite rule then we use that rule to simplify all existing axioms. This process terminates since each rewriting application reduces the length of the terms of the axioms that it rewrites, which means that in general, the addition of an axiom can only lead to a finite number of rewriting and elimination steps.

We now sketch how to rewrite the example Axioms (4) and (5) as shown above. Suppose that Axiom (4) already exists and we are about to add Axiom (5). First, we try to rewrite Axiom (5) using Axiom (4) as a rewriting rule. Unfortunately since the left (longer) term of Axiom (5) does not unify with any subterm in Axiom (4) rewriting fails. We find that Axiom (5) does not already exist and it is not a trivial axiom so we add it to the set of known axioms. Since Axiom (5) is a rewriting rule, we try to rewrite all existing axioms, namely Axiom (4). We find that the left side of Axiom (5) unifies with the following subterm of Axiom (4)

$$\mathit{pop}(\mathit{push}(\mathit{push}(x_4, i_4).\mathit{state}, j_4).\mathit{state}).\mathit{state}$$

with the unifier

$$\{x_5 \rightarrow push(x_4, i_4).state,\ i_5 \rightarrow j_4\} \quad .$$

Therefore, we instantiate the right side of Axiom (5) with the unifier we found and obtain

$$push(x_4, i_4).state$$

which we use as a replacement for the subterm that we found in Axiom (4). Therefore, Axiom (4) rewrites to

$$\forall x_4 :\ ObjectStack, \forall i_4 : Object\ (\ pop(push(x_4, i_4).state).state = x_4\ ) \qquad (6)$$

which is equivalent to Axiom (5). Since the rewritten Axiom (4) is identical to Axiom (5), we eliminate Axiom (4). In summary, we end up with Axiom (5) as the only axiom in the final set of axioms.

3.7 Discussion

The system we described in this section can discover algebraic specifications from Java classes. The system is dynamic, which means that the output is potentially unsound (i.e., may discover incorrect axioms) or incomplete (i.e., may miss some axioms) or both. In our experience, we can usually achieve soundness by increasing the maximum size for the generated terms but that is costly. Our system has a bias towards certain patterns of equations and axioms; one can add new patterns if desired.

The main goal of the tool is to provide a good starting point in documenting code; the tool is successful as long as it reduces the time and effort for documentation.

4 Evaluation

This section evaluates our ideas. Section 4.1 uses various metrics to characterize the performance of our algebraic specification discovery tool. Section 4.2 discusses an example where an automatically detected specification is used to simulate the behavior of a class for an existing program. Section 4.3 then gives an example in which a manually generated specification is compared to an automatically generated one.

4.1 Performance Evaluation of our Algebraic Specification Discovery Tool

We conducted our evaluations on a Dell PowerEdge 600SC Pentium 4 2.4 GHz single-CPU workstation with 2 GB of RAM running SuSE Linux 9.0 and Sun JDK 1.5.0.

Our discovery mechanism is parameterized by *maximum term size*, the maximum number of variables, constants and operations (ignoring *state* and *retval*

annotations) allowed per term. For example, the size of

$$push(IntStack().state, 4).state$$

is 3.

To test the efficiency of axiom discovery, we configured our system as follows: As a default, we used a term size of 5 for the equation generators and a test case size of four. This means that the resulting equations will have terms with a maximum size of five on either side, and that they will be tested with test stubs (see Algorithm 3.3.4) of size four or smaller.

For `HashMap`, we chose a term size of 4 for all equation generators except for the observer equation generator, where a size of 6 was beneficial. We configured the system for `HashSet` in the same way as `HashMap`, except that we found that a test stub size of 3 was sufficient. We also configured three distinct instances of `Object` and similar small pools for primitive types.

Table 1 describes the our benchmark programs. Column "# op" gives the number of operations in the class and "# observ" gives the number of operations that are observers (i.e., operations with a non-void return type). `LinkedList`, `HashSet`, `HashMap`, and `Hashtable` are library classes from Sun's Java Development Kit 1.5.0.

Table 1: Java classes used in our evaluation

Java Class	**Description**	Source	# op	# observ.
IntegerStack	minimal integer stack	Henkel	13	6
IntegerStack2	integer stack	Henkel	13	6
ObjectStack	object stack (Fig. 2)	Henkel	12	6
FullObjectStack	another Object stack	Henkel	16	9
IntegerQueue	a FIFO queue for integers	Henkel	16	9
ObjectMapping	a mapping between objects	Henkel	16	10
ObjectQueue	a FIFO queue for objects	Henkel	16	9
LinkedList	linked list	Sun	46	37
HashSet	hash set	Sun	27	21
HashMap	hash map	Sun	26	19
Hashtable	hash table	Sun	29	22

Section 4.1.1 gives performance characteristics for our system. Section 4.1.2 presents data that suggests that our tool is successful in exercising most of the class under consideration and is thus likely to be mostly sound. Finally Section 4.1.3 discusses some axioms that our tool discovers.

4.1.1 Performance of the Tool

Table 2 gives the overall performance of our system. For each benchmark, we display the number of axioms before and after rewriting and the time it took to generate the axioms. The table shows that our redundancy reduction by rewriting

Table 2: Timings for our benchmark programs.

benchmark	# axioms before rewriting	# final axioms	time
IntegerStack	71	7	3.5s
IntegerStack2	81	9	2.5s
ObjectStack	73	7	5.3s
FullObjectStack	198	19	7.9s
IntegerQueue	298	22	31.8s
ObjectQueue	276	17	56.5s
ObjectMapping	184	18	6.6s
LinkedList	1478	113	9.1 min
HashSet	4959	55	22.9 min
HashMap	5795	60	191 min
Hashtable	5702	44	167 min

Table 3: Efficiency of Term generation.

algebra	time in seconds (# generated terms)			
		size 10		size 11
IntegerStack	0.28	(166)	0.52	(418)
IntegerStack2	0.25	(251)	0.42	(566)
ObjectStack	0.71	(432)	1.35	(1472)
FullObjectStack	0.99	(432)	1.75	(1472)
IntegerQueue	0.70	(358)	1.23	(722)
ObjectQueue	1.62	(906)	3.17	(2271)
ObjectMapping	1.05	(441)	1.58	(441)
LinkedList	22.27	(2616)	55.65	(5108)
HashSet	54.47	(1732)	121.09	(2390)
HashMap	24.57	(7966)	49.08	(13981)
Hashtable	12.57	(3423)	22.10	(5158)

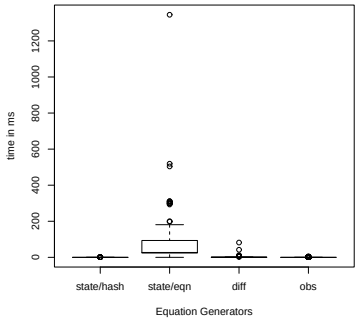


Figure 6: Efficiency of equation generation (`LinkedList`).

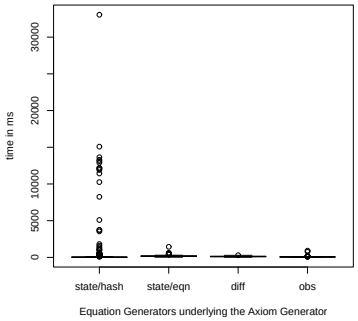


Figure 7: Efficiency of axiom generation (`LinkedList`).

is very effective. It also shows that our system is fast for all of our own test cases, and reasonably efficient for `LinkedList` and `HashSet`.

We see from Table 2 that `LinkedList` has the largest number of axioms– 113. While this is a significant number, it is worth noting that the Java standard implementation of linked list has a large number of operations (46, if we include all public operations defined by its superclasses) and thus, 113 axioms is not excessive.

Table 3, Figure 6, Figure 7, and Figure 8 explore the performance of our system in detail. Figure 3 gives the time to generate all terms of sizes 10 (first column) and 11 (second column) respectively. We see that the number of terms does increase significantly with term length, even though we prune away many useless terms. We also see that classes with a large number of terms (e.g., `HashSet`) are the ones that take the most time in our system[2].

Figure 6 and 7 are box plots that give the distribution of the time to generate the different kinds of equations and axioms for `LinkedList`. As an example, consider the state/eqn plot in Figure 6. The box denotes the interquartile range, which contains the 50% of the values that fall within the second and third quartile. The line inside the box is the median, which overlaps with the bottom of the box due to a strong bias towards low values. Values beyond the whiskers are outliers. From Figure 6 and Figure 7 we see that the state axioms and equations are the most expensive to generate with state/eqn being the slowest. For the most part, observer and difference equations and axioms are fast to generate and all equations and axioms of those type take approximately the same time. These results suggest that it may be worthwhile to try other orderings or kinds of equations in order to speed up the equation and axiom generation.

[2]Note that these results differ from the ones reported in [HD03] since we made many modifications to our system since our earlier paper.

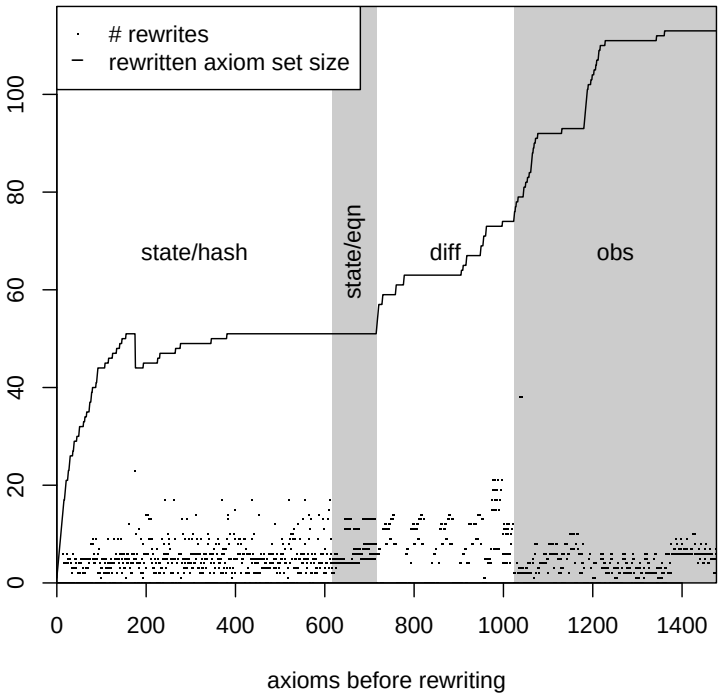


Figure 8: Learning curve (`LinkedList`)

Fig. 8 gives insight into the behavior of our tool when discovering axioms for `LinkedList`. The x axis denotes time in terms of axioms generated by the axiom generator. The "learning curve" is the number of axioms that have been discovered and rewritten. Note that this curve does not ascend monotonically, as the discovery of one axiom can lead to the elimination of numerous other axioms. The dots near the bottom denote the number of rewriting events per discovered axiom. We run our system as follows: first, we discover state axioms using the hash table optimization (state/hash), then we generate the general state axioms (state/eqn), next we generate the difference axioms and finally the observer axioms. The shaded areas in Fig. 8 denote these different zones.

Note the sudden dip around axiom 176. At this point, we discover the following:

```
forall x0:LinkedList
  LinkedList(clear(x0).state).state == LinkedList().state
```

Together with the earlier axiom

```
forall x0:LinkedList
  LinkedList(x0).state == x0
```

this allows our system to eliminate 8 axioms that examined the behavior of "cleared" LinkedLists (which our system then reduced to the behavior of newly constructed LinkedLists).

Further note that State/eqn generates only 100 out of the axioms considered here, many of which either were comparatively simple (resulting in few generalization steps, cf. Algorithm 6), could be simplified by axioms previously detected by state/hash, or could be falsified quickly due to a cache we provided for test parameters that had previously been successful in refuting axiom candidates. Thus, despite having to employ the (comparatively) expensive observational equivalence mechanism (Section 3.3), the total run-time of state/eqn only amounts to about twenty seconds of the 9.1 minutes (average) required to determine the specification of `LinkedList`: Its increased cost in equation generation was shadowed by its significantly decreased cost of axiom generation.

4.1.2 Coverage Measurements

Since our system is based on a dynamic approach it is neither sound nor complete. One way in which our system can fail to be sound is if the generated terms do not adequately exercise the full range of behavior of the class under consideration. Table 4 gives the basic block coverage attained by our term generator. Overall, we find that our terms yield a high coverage.

When our coverage is less than 100% it is often because our terms do not adequately explore the argument space for operations (e.g., for *LinkedList* we did not generate a term that called `removeAll` with a collection that contains a subset of the receiver's elements). Other reasons include: dead code (e.g., some code in `LinkedList.subList`); corner cases (e.g., corner cases for `initialCapacity` in `Hashtable`).

Table 4: Basic block coverage.

Class	Coverage
`IntegerStack`	100%
`IntegerStack2`	62.5%
`ObjectStack`	100%
`FullObjectStack`	100%
`IntegerQueue`	100%
`ObjectQueue`	100%
`ObjectMapping`	100%
`LinkedList`	85.7%
`HashSet`	85%
`HashMap`	66.7%
`Hashtable`	74.8%

Table 5: Fraction of correct axioms (based on manual inspection).

Class	Correct/Total Axioms	Precision
`IntegerStack`	6/7	85.7%
`IntegerStack2`	7/9	77.8%
`ObjectStack`	6/7	85.7%
`FullObjectStack`	18/19	94.7%
`IntegerQueue`	21/22	95.5%
`ObjectQueue`	16/17	94.1%
`ObjectMapping`	17/18	94.4%
`LinkedList`	109/113	96.5%
`HashSet`	53/55	96.4%
`HashMap`	59/60	98.3%
`Hashtable`	43/44	97.7%

4.1.3 Manual Inspection of Axioms

To make sure that the axioms discovered by our tool were correct, we manually verified the generated axioms for all classes; the results are listed as "Precision" in Table 5.

The set of incorrect axioms we encountered typically included the incorrect equality axiom discussed at the end of Section 4.3.

The axioms though many for some classes (e.g., *LinkedList*), were relatively easy to read (verifying them took half an hour of our time per class, at most).

We now discuss a number of representative sample axioms which were discovered by our tool.

```
forall x0:HashMap
  size(x0).state == x0                                    (Axiom 3)
```

Axiom 3 says that invoking *size* on a *HashMap* does not modify its internal state. Our system generates such axioms for each pure observer (i.e., an operation that has a non-void return value and does not change the state of the object), for example, it finds 16 such axioms for *LinkedList*.

```
forall x0:HashMap                                          (Axiom 4)
   putAll(x0,x0).state == x0
```

Axiom 4 says that if we add all mappings of a *HashMap* into an equivalent *HashMap*, the receiver's state does not change.

```
forall x0 : HashMap                                        (Axiom 5)
forall x1 : Object
forall x2 : Object
   get(put(x0, x1, x2).state, x1).retval == x2
```

Axiom 5 gives a partial characterization of the *get* and *put* operations.

```
forall x0 : Object                                         (Axiom 6)
   get(put(HashMap().state, Object@0, x0).state, Object@1).retval
   == null
```

Axiom 6 is one of the axioms that could be more abstract if our tool would support conditional axioms: Instead of using the constants Object@0 and Object@1, the axiom could then be rewritten into

```
forall x0 : Object
forall x1 : Object
forall x2 : Object
if x1.equals(x2) == false then
   get(put(HashMap().state, x1, x0).state, x2).retval == null
```

As these kinds of overly concrete axioms require post-processing to be useful, we removed them for all of our statistical results (i.e., they were omitted in all of our graphs and tables).

Finally, it is worth noting that while we can manually determine the ratio of correct axioms generated by our tool (its *precision*), it is much harder to determine how many axioms we are missing (to determine *recall*). One approach to approximate an answer to this question is to run our algebraic specification interpreter for a particular unit-test or application, and add algebraic axioms to the specification until the interpretation succeeds. We describe such a scenario in Section 4.2.

4.2 Case Study: Discovering a Specification for Interpretation

In this case study, we used our specification discovery tool to generate a specification for the `java.util.ArrayList` class contained in Sun's Java Development Kit. We then debugged this specification with our algebraic interpreter [HD04b], simulating

the `ArrayList` class for a BibTeX parser.[3] We chose this client application because it is not dependent on libraries other than the Java standard libraries, it uses collection classes, and because we were familiar with its implementation.

Out of the 10 algebraic axioms to execute the BibTeX parser successfully, our discovery tool can produce 3 axioms exactly as needed. As an example, the following two axioms specify how the first element of an `ArrayList` can be obtained by applying the `get` operation for index 0:

```
forall x0 : Object                                              (Axiom 7)
  get(add(newArrayList().state, x0).state, 0).retval == x0

forall l : ArrayList                                            (Axiom 8)
forall o1 : Object
forall o2 : Object
  get(add(add(l, o1).state, o2).state, 0).retval
  == get(add(l, o1).state, 0).retval
```

We manually added 7 axioms to the specification. Five of those axioms describe the behavior of `Iterator` instances generated by `ArrayList` objects. For example, the following axiom states that an iterator created from an empty list does not have a next element:

```
hasNext(iterator(ArrayList().state).retval).retval == false
```

The discovery tool currently cannot find these 5 axioms because the state of the `Iterator` object is modeled as the return value of the operation `iterator()` of another class (`ArrayList`). This scenario is not covered by the currently implemented equation generators. However, the discovery tool provides extension points for adding new equation generators. An appropriate equation generator could be implemented without changing the infrastructure.

More details about this case study, including all discovered axioms, are available in a technical report [HD04a]. While this case study was performed with an older version of our tool, we verified that our previous observations still hold with the current version.

4.3 Case Study: Comparing a Discovered Specification to a Hand-Written one

In this case study, we compared a discovered specification for a priority queue class to a specification of the same class that we had furnished beforehand. This specification was assumed to be "complete", i.e., we had already gained reasonable confidence in it specifying all aspects we considered relevant. Unlike the previous case study, our goal was to determine whether the specification would be sufficient for all possible clients, not just for one run of one particular program.

Figure 9 gives the class signature of this queue. In addition to the methods given in Figure 9, the `PriorityQueue` also inherits the `toString()`, `hashCode()`,

[3] Available at `www.cs.colorado.edu/~henkel/stuff/javabib/`.

```
public class PriorityQueue {
       public PriorityQueue(Comparator c){...}
       public Object get(){...}
       public boolean equals(Object other){...}
       public boolean add(Object object){...}
       public boolean addAll(PriorityQueue collection){...}
       public boolean contains(Object object){...}
       public int size(){...}
}
```

Figure 9: A priority queue for Java

and `getClass()` methods from `Object`. The constructor of the `PriorityQueue` takes an instance of `java.util.Comparator`, which determines the preorder on the queue's values.

4.4 Comparing a hand-crafted specification to an automatically detected one

Using our algebraic specification interpreter [HD04b] and a test suite consisting of 92 run-time tests, we developed what we considered to be a complete algebraic priority queue specification. In order to function properly with our interpreter, this specification also included a *hidden method*, describing an auxiliary function.

We then wrote a Java implementation of a Priority Queue and validated it dynamically by running it against the test suite and against the interpreted specification.

4.4.1 Detected axioms vs. specified axioms

Our discovery tool yielded 28 axioms in 11 seconds for the Java Priority Queue.

- 8 axioms exactly matched axioms we had specified by hand, modulo renaming.
- 6 axioms matched axioms we had specified by hand, except that they were slightly more concrete.
- 6 axioms were correct but could have been re-written if certain other, missing axioms had also been known.
- 3 axioms specified the semantics of methods we had inherited but chose not to specify (`toString`, `getClass` and `hashCode`).
- 2 axioms were correct but could have been eliminated by applying basic laws of arithmetic.
- 2 axioms specified relevant behavior we had omitted in our manual specification.

- 1 axiom was incorrect

The 6 axioms listed as "could have been rewritten if [...] missing axioms had also been known" were found because our current detection mechanism does not attempt to derive conditional axioms (axioms with preconditions).

The 6 axioms described as "matched axioms [...], except that they were slightly more concrete" were axioms in which the default constructor was used, even though the more general parameterized constructor would have allowed for them to be generalized over all supported orders; this was caused by an overly restrictive size of our test set for `java.util.Comparator` objects.

4.4.2 Incorrectly detected axioms

As mentioned above, we detected an incorrect axiom, namely the following:

```
forall x0 : edu.colorado.cs.PriorityQueue                    (Axiom 9)
forall x1 : java.lang.Object
  equals(x0,x1).retval == false
```

Our dynamic testing does not currently consider subtyping when choosing test values. As such, we never attempted to pass a priority queue to `equals` and thus never encountered a situation in which `equals` would evaluate to `true`, explaining Axiom 9.

4.4.3 Specified axioms vs. detected axioms

Out of the 30 axioms we originally specified:

- 8 axioms were detected (modulo renaming).
- 7 axioms required preconditions and therefore could not be detected.
- 6 axioms were detected, but used the (overly concrete) default constructor (see above).
- 5 axioms were not detected because our system does not consider subtype instances as valid test parameters.
- 2 axioms specified a hidden method.
- 1 axiom was not detected because `null` is not used as a test parameter in places requiring an instance of the class currently being analyzed.
- 1 axiom was not detected because none of our equation generators emitted an equation that could have served as a template for it.

This last axiom was the following:

```
forall p : edu.colorado.cs.PriorityQueue
forall q : edu.colorado.cs.PriorityQueue
forall o : java.lang.Object
   addAll(p, add(q, o).state).state
   == add(addAll(p, q).state, o).state
```

None of our equation generators yielded an equation that could have been generalized to this axiom. Again (as mentioned in section 4.2), it should be noted that such a generator can be added easily; we did not include it in our system because our initial estimates had indicated that the space of equations we would have to explore would be quite significant, slowing down discovery, while only very few additional axioms could potentially be found.

5 Related Work

We now describe related work in specification languages, dynamic invariant detection, automatic programming, static analysis, and testing.

5.1 Specification Languages

We drew many ideas and inspirations from previous work in algebraic specifications for abstract data types [GH78]. Horebeek and Lewi [HL89] give a good introduction to algebraic specifications. Sannella *et al.* give an overview of and motivation for the theory behind algebraic specifications [ST97]. A book by Astesiano *et al.* contains reports of recent developments in the algebraic specification community [AKKB99].

Antoy describes how to systematically design algebraic specifications [Ant89]. In particular, he describes techniques that can help to identify whether a specification is complete. His observations could be used in our setting; however, they are limited to a particular class of algebras.

Prior work demonstrates that algebraic specifications are useful for a variety of tasks. Rugaber *et al.* study the adequacy of algebraic specifications for a reengineering task [RSS01]. They specified an existing system using algebraic specifications and were able to regenerate the system from the specifications using a code generator. Janicki *et al.* find that for defining abstract data types, algebraic specifications are preferable over the trace assertion method [BP78, JS01]

5.2 Dynamic Invariant Detection

Recently, there has been much work on dynamic invariant detection [ECGN01, WML02, ABL02, HL02]. Dynamic invariant detection systems discover specifications by learning general properties of a program's execution from a set of program runs.

Daikon [ECGN01] discovers Hoare-style axiomatic specifications [Hoa69, Gri81]. Daikon is useful for understanding *how* something is implemented, but also exposes the full complexity of a given implementation. Daikon has been improved in many

ways [ECGN00, EGKN00, DDLE02] and has been used for various applications including program evolution [ECGN01], refactoring [KEGN01], test suite quality evaluation [HME], bug detection [HL02], and as a generator of specifications that are then checked statically [NE02].

Whaley *et al.* [WML02] describe how to discover specifications that are finite state machines describing in which order method calls can be made to a given object. Similarly, Ammons *et al.* extract nondeterministic finite state automatons (NFAs) that model temporal and data dependencies in APIs from C code [ABL02]. These specifications are not nearly as expressive as algebraic specifications, since they cannot capture what values are returned by the methods.

Our preliminary studies show that the current implementation of our tool does not scale as well as some of the systems mentioned above. However, we are unaware of any dynamic tool that discovers high-level specifications of the interfaces of classes. Also, unlike prior work, our system interleaves automatic test generation and specification discovery. All previous systems require a test suite.

5.3 Automatic Programming

Automatic programming systems [Bie72, BK76, Bie78, Ang82, Har75, Sum77] discover programs from examples or synthesize programs from specifications by deduction. The programs are analogous to our specifications in that our specifications are high-level descriptions of examples. Algorithmic program debugging [Sha82] is similar to automatic programming and uses an inductive inference procedure to test side-effect and loop-free programs based on input output examples and then helps users to interactively correct the bugs in the program. Unlike the above techniques whose goals are to generate programs or find bugs, the goal of our system is to generate formal specifications (which could, of course, be used to generate programs or find bugs).

5.4 Static Analysis

Program analyses generate output that describes the behavior of the program. For example, shape analyses [SRW02] describe the shape of data structures and may be useful for debugging. Type inference systems, such as Lackwit [OJ97] generate types that describe the flow of values in a program. Our system is a dynamic black-box technique that does not need to look at the code to be effective. However, various static techniques can be used to guide our system (for example, we have already experimented with mod-ref analyses).

5.5 Testing

Woodward describes a methodology for *mutation testing* algebraic specifications [Woo93]. Mutation testing introduces one change ("mutations") to a specification to check the coverage of a test set. Woodward's system includes a simple test generation method that uses the signatures of specifications.

Algebraic specifications have been used successfully to test implementations of abstract data types [GMH81, San91, HS96, AH00]. One of the more recent systems, Daistish [HS96] allows for the algebraic testing of OO programs in the presence of side effects. In Daistish, the user defines a mapping between an algebraic specification and an implementation of the specification. The system then checks whether the axioms hold, given user-defined test vectors. Similarly, the system by Antoy *et al.* [AH00] requires the users to give explicit mappings between specification and implementation. Our system automatically generates both the mapping and the test suite.

Doong and Frankl [DF94] introduce the notion of observational equivalence and an algorithm that generates test cases from algebraic specifications. Their system semi-automatically checks implementations against generated test cases. Later work improved on Doong and Frankl's test case generation mechanism [CTCC98] by combining white-box and black-box techniques. Our tool can potentially benefit from employing static analysis of the code when generating test cases (white box testing).

In addition to the above, prior work on test-case generation [HHG90, BOP00, MOP02] is also relevant to our work, particularly where it deals with term generation. Also, methods for evaluating test suites or test selection [ZHM97, GHK$^+$01] are relevant. We do not use these techniques yet but expect that they will be useful in improving the speed of our tool and the quality of the axioms.

Korat is a system for automated testing of Java programs [BKM02]. Korat translates a given method's pre- and post-conditions into Java predicates, generates an exhaustive set of test cases within a finite domain using the pre-condition predicate and checks the correctness of the method by applying the post-condition predicate. Our approach for generating terms borrows ideas from Korat.

6 Conclusion

Java applications rely heavily on reusable container classes. In our study of a number of large Java applications we found that up to 30% of the classes in our applications used one or more of the containers in the `java.util` package. Thus, it is important for the container classes to be well documented.

To assist in this task, we describe a specification discovery tool that automatically discovers algebraic specifications for Java classes. Since algebraic specifications provide a clear, concise, and ambiguous description of the interface of container classes, they are ideal for documenting these classes.

The specification discovery tool works by observing the runtime-behavior of objects. Our experiments with a number of Java classes reveal that our system generates axioms that are both correct and useful for understanding and using container classes. Since our discovery tool is a dynamic tool (i.e., based on observing runtime behavior), it is neither sound nor complete. However, the discovered specifications give developers a good starting point for developing complete and sound specifications.

Acknowledgements

We thank the members of CU Boulder's programming languages group, the members of CU Boulder's software engineering research laboratory, members of the software technology department at IBM Research, and the anonymous referees for POPL and ECOOP for listening to our ideas and giving great feedback.

References

[ABL02] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[AH00] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1), January 2000.

[AKKB99] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of Systems Specification*. Springer, 1999.

[Ang82] Dana Angluin. Inference of reversible languages. *Journal of the ACM (JACM)*, 29(3):741–765, 1982.

[Ant89] S. Antoy. Systematic design of algebraic specifications. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, Pittsburgh, Pennsylvania, 1989.

[Bie72] A. W. Biermann. On the inference of turing machines from sample computations. *Artificial Intelligence*, 3:181–198, 1972.

[Bie78] Alan W. Biermann. The inference of regular Lisp programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics*, 8:585–600, August 1978.

[BK76] Alan W. Biermann and Ramachandran Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering*, 2(3):141–153, September 1976.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *International Symposium on Software Testing and Analysis*, Rome, Italy, July 2002.

[BOP00] U. Buy, A. Orso, and M. Pezze. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, Portland, Oregon, 2000.

[BP78] W. Bartussek and D. L. Parnas. Using assertions about traces to write abstract specifications for software modules. In *Information Systems*

Methodology: Proceedings, 2nd Conference of the European Cooperation in Informatics, Venice, October 1978; Lecture Notes in Computer Science, volume 65. Springer Verlag, 1978.

[CTCC98] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object oriented programs. *ACM Transactions on Software Engineering*, 7(3), July 1998.

[DDLE02] Nii Dodoo, Alan Donovan, Lee Lin, and Michael D. Ernst. Selecting predicates for implications in program analysis. http://pag.lcs.mit.edu/~mernst/pubs/invariants-implications.pdf, March 2002.

[DF94] R. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering*, 3(2), April 1994.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, June 2000.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *ACM Transactions on Software Engineering*, 27(2):1–25, February 2001.

[EGKN00] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. Dynamically discovering program invariants involving collections. TR UW-CSE-99-11-02, University of Washington, 2000. revised version of March 17, 2000.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, 2000.

[GH78] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GHK+01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2):184–208, 2001.

[GMH81] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, 1981.

[Gri81] David Gries. *The science of programming*. Texts and monographs in computer science. Springer-Verlag, 1981.

[Har75] Steven Hardy. Synthesis of Lisp functions from examples. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 240–245, 1975.

[HD03] Johannes Henkel and Amer Diwan. Discovering algebraic specifications from Java classes. In Luca Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, Darmstadt, July 2003. Springer.

[HD04a] Johannes Henkel and Amer Diwan. Case study: Debugging a discovered specification for java.util.arraylist by using algebraic interpretation. Technical Report CU-CS-970-04, University of Colorado at Boulder, 2004.

[HD04b] Johannes Henkel and Amer Diwan. A tool for writing and debugging algebraic specifications. In *International Conference on Software Engineering (ICSE)*, 2004.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 169–180. ACM Press, 1990.

[HL89] Ivo Van Horebeek and Johan Lewi. *Algebraic specifications in software engineering: an introduction*. Springer-Verlag, 1989.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, May 2002.

[HME] Michael Harder, Benjamin Morse, and Michael D. Ernst. Specification coverage as a measure of test suite quality. Septermber 25, 2001.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HS96] M. Hughes and D. Stotts. Daistish: Systematic algebraic testing for OO programs in the presence of side-effects. In *Proceedings of the International Symposium on Software Testing and Verification*, San Diego, California, 1996.

[JS01] R. Janicki and E. Sekerinski. Foundations of the trace assertion method of module interface specification. *ACM Transactions on Software Engineering*, 27(7), July 2001.

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *International Conference on Software Maintenance*, Florence, Italy, 2001.

[Mit96] John C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[MOP02] Vincenco Martena, Alessandro Orso, and Mauro Pezze. Interclass testing of object oriented software. In *Proc. of the IEEE International Conference on Engineering of Complex Computer Systems*, 2002.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA), Rome*, July 2002.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *International Conference on Software Engineering*, pages 338–348, 1997.

[RSS01] Spencer Rugaber, Terry Shikano, and R. E. Kurt Stirewalt. Adequate reverse engineering. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering*, pages 232–241, 2001.

[San91] S. Sankar. Run-time consistency checking of algebraic specifications. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, Victoria, British Columbia, Canada, September 1991.

[Sha82] Ehud Y. Shapiro. *Algorithmic Program Debugging*. ACM Distinguished Dissertation 1982. MIT Press, 1982.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[ST97] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[Sum77] Phillip D. Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

[WML02] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium of Software Testing and Analysis*, 2002.

[Woo93] M. R. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. *IEEE Software Engineering Journal*, 8(4):237–245, July 1993.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.